

## Practically Groovy: Stir some Groovy into your Java apps

Exploit Groovy's simplicity by embedding simple, easy-to-write scripts

Skill Level: Intermediate

[Andrew Glover \(andrew@thirstyhead.com\)](mailto:andrew@thirstyhead.com)

Co-Founder

ThirstyHead.com

[Scott Davis \(scott@thirstyhead.com\)](mailto:scott@thirstyhead.com)

Founder

ThirstyHead.com

24 May 2005

Ever thought about embedding Groovy's simple, easy-to-write scripts in your more complex Java™ programs? In this installment of *Practically Groovy*, Andrew Glover shows you the many ways to incorporate Groovy into your Java code and explains where and when it's appropriate to do so.

If you've been reading this series for a while, you've seen that there are all sorts of interesting ways to use Groovy, and that one of the chief advantages of Groovy is its productivity. Groovy code is often easier and faster to write than Java code, which makes it a worthwhile addition to your development toolkit. On the other hand, as I have stressed over and over again in this series, Groovy isn't -- and isn't intended to be -- a replacement for the Java language. So the question is, can you incorporate Groovy into your Java programming practice, and is it, or *when is it*, useful to do so?

This month, I'll try to answer that question. I'll start with something familiar -- how Groovy scripts get compiled into Java-compatible class files -- and then delve further under the hood to see exactly how the Groovy compilation facility (`groovyc`) makes this magic happen. Understanding what's going on under the hood with Groovy is the first step to using it in your Java code

Note that some of the programming techniques demonstrated in this month's examples are at the core of the `Groovlets` framework and Groovy's `GroovyTestCase`, which I've discussed in previous articles.

### About this series

The key to incorporating any tool into your development practice is knowing when to use it and when to leave it in the box. Scripting languages can be an extremely powerful addition to your toolkit, but only when applied properly to appropriate scenarios. To that end, [Practically Groovy](#) is a series of articles dedicated to exploring the practical uses of Groovy, and teaching you when and how to apply them successfully.

## A marriage made in heaven?

Earlier in this series, when I showed you how to [unit test normal Java programs with Groovy](#), you may have noticed something peculiar: I *compiled* those Groovy scripts. In fact, I compiled my groovy unit tests into normal Java `.class` files and ran them as part of a Maven build.

This type of compilation is done by invoking the `groovyc` command, which compiles Groovy scripts into plain old Java-compatible `.class` files. For example, if your script declares three classes, calling `groovyc` will result in at least three `.class` files. The files themselves will follow the standard Java rules where `.class` file names match declared class names.

As an example, take a look at Listing 1, which creates a simple script that declares a few classes. You can then see for yourself what the `groovyc` command generates:

### Listing 1. Class declaration and compilation in Groovy

```
class Person {
    String fname
    String lname
    int age
    String address
    List contactNumbers = []

    String toString(){
        def numstr = new StringBuffer()
        if (contactNumbers != null){
            contactNumbers.each{
                numstr.append(it)
                numstr.append(" ")
            }
        }
        return "first name: ${fname} " +
            "last name: ${lname} " +
            "age: ${age} " +
            "address: ${address} " +
            "contact numbers: ${numstr.toString()}"
    }
}
```

```

    }
}

class Address {
    String street1
    String street2
    String city
    String state
    String zip

    String toString(){
        return "street1: ${street1} " +
            "street2: ${street2} " +
            "city: ${city} " +
            "state: ${state} " +
            "zip: ${zip}"
    }
}

class ContactNumber {
    String type
    String number

    String toString(){
        return "Type: ${type} " +
            "number: ${number}"
    }
}

def nums = [new ContactNumber(type:"cell", number:"555.555.9999"),
            new ContactNumber(type:"office", number:"555.555.5598")]
def addr = new Address(street1:"89 Main St.", street2:"Apt #2",
                      city:"Utopia", state:"VA", zip:"34254")
def pers = new Person(fname:"Mollie", lname:"Smith", age:34,
                     address:addr, contactNumbers:nums)
println pers.toString()

```

In Listing 1, I declared three classes -- Person, Address, and ContactNumber. The code that follows creates objects of the newly defined types and calls a `toString()` method. Pretty simple stuff so far, but now look at what `groovyc` generated as a result, in Listing 2:

## Listing 2. Classes generated by the `groovyc` command

```

aglover@l2d21 /cygdrive/c/dev/project/target/classes/com/vanward/groovy
$ ls -ls
total 15
 4 -rwxrwxrwx+ 1 aglover  user   3317 May  3 21:12 Address.class
 3 -rwxrwxrwx+ 1 aglover  user   3061 May  3 21:12 BusinessObjects.class
 3 -rwxrwxrwx+ 1 aglover  user   2815 May  3 21:12 ContactNumber.class
 1 -rwxrwxrwx+ 1 aglover  user   1003 May  3 21:12
  Person$_toString_closure1.class
 4 -rwxrwxrwx+ 1 aglover  user   4055 May  3 21:12 Person.class

```

Wow, *five* .class files! The Person, Address, and ContactNumber files make sense, but what about the other two?

It turns out that the `Person$_toString_closure1.class` is the result of the closure found in the Person class's `toString()` method. It is internally an inner

class of `Person`. But what about the `BusinessObjects.class` file -- what could it be?

A closer look at [Listing 1](#) reveals that the code I wrote in the main body of the script, after I declared those three classes, became a `.class` file, which is named after the name of the script. In this case, the script was named `BusinessObjects.groovy`, so the code not contained in a class definition was then compiled into a `.class` file named `BusinessObjects`.

## Coding in reverse

Decompiling these classes can be quite interesting. The resultant `.java` files can be rather large because of the nature of Groovy's under-the-covers code; however, what you will notice is the difference between classes declared in a Groovy script (like `Person`) and the code that lives outside of classes (like that found in `BusinessObjects.class`). Classes defined in Groovy files end up implementing `GroovyObject` and code defined outside of a class is bundled into a class that extends `Script`.

For example, if you examine the resultant `.java` file of `BusinessObjects.class`, you'll find it defines a `main()` method and a `run()` method. Not surprisingly, the `run()` method contains the code I wrote to create new instances of those objects and the `main()` method calls the `run()` method.

Again, the point of all this detail is that the better you understand Groovy, the easier it will be for you to incorporate it into your Java programs. "And why would I want to do that," you ask? Well, let's just say you developed something cool in Groovy; wouldn't it be nice to be able to incorporate it into your Java programs, too?

Just for the sake of argument, I'll first *attempt to* create something useful in Groovy; then, I can explore the various ways it could be embedded in a normal Java program.

## Making music Groovy again

I love music. In fact, my CD collection rivals my computer books collection. Over the years, I've ripped my music onto various computers and, in the process, muddled my MP3 collection to the point where I have numerous directories representing a cornucopia of music.

Recently, I took a first step toward getting my music collection back in order. I wrote a quick Groovy script to iterate over a collection of MP3 files in a directory and give me detailed information about each file, such as artist, album title, etc. The script is shown in [Listing 3](#):

### Listing 3. A very useful Groovy script

```
package com.vanward.groovy

import org.farng.mp3.MP3File
import groovy.util.AntBuilder

class Song {
    def mp3file

    Song(String mp3name){
        mp3file = new MP3File(mp3name)
    }

    String getTitle(){
        mp3file.getID3v1Tag().title
    }

    String getAlbum(){
        mp3file.getID3v1Tag().album
    }

    String getArtist(){
        mp3file.getID3v1Tag().artist
    }

    String toString(){
        return "Artist: ${getArtist()} " +
            "Album: ${getAlbum()} " +
            "Song: ${getTitle()}"
    }

    static List getSongsForDirectory(String sdir){
        println "sdir is: ${sdir}"
        def ant = new AntBuilder()
        def scanner = ant.fileScanner {
            fileset(dir:sdir) {
                include(name:"**/*.mp3")
            }
        }
        def songs = []
        scanner.each{ f ->
            songs << new Song(f.getAbsolutePath())
        }
        return songs
    }
}

def songs = Song.getSongsForDirectory(args[0])
songs.each{
    println it
}
```

As you can see, the script is fairly simple, especially for being so useful to someone in my shoes. All I have to do is pass in a particular directory name and I'm given the relevant information (artist name, song name, and album) for every MP3 file in the directory.

Now let's see what I need to do to incorporate this nifty script into a normal Java program that could organize music via a database or even play MP3s.

## Class files are class files

As discussed earlier, my first option would be to simply compile the script using `groovyc`. In this case, I expect `groovyc` would create *at least* two `.class` files -- one for my `Song` class and another for the script code following `Song`'s declaration.

In fact, `groovyc` would generate five `.class` files. This correlates to the fact that `Songs.groovy` contains three closures, two in the `getSongsForDirectory()` method and one in the script body, where I iterated over the collection of `Songs` and called `println`.

Because three of the `.class` files are really inner classes of `Song.class` and `Songs.class`, I only need to focus on two `.class` files. `Song.class` maps directly to the `Song` declaration in the Groovy script and implements `GroovyObject`, while `Songs.class` represents the script code after I defined `Song`, and therefore extends `Script`.

At this point, I have two options for how I will incorporate my newly compiled Groovy code into Java code: I could actually run the code via the `main()` method found in the `Songs.class` file (because it extends `Script`), or I could include the `Song.class` in my classpath and use it as I would any other object in my Java code.

## Take it easier

Calling the `Songs.class` file via the `java` command is painfully simple, as long as you remember to include Groovy's associated dependencies and any dependencies your Groovy script may have required. The easiest way to include Groovy's required classes is to insert the all-in-one Groovy-embeddable jar file in your classpath. In my case, the file is `groovy-all-1.5.7.jar`. To run `Songs.class`, I'll also need to remember to include the MP3 library I used (`jd3lib-0.5.jar`). Listing 4 puts it together:

### Listing 4. Groovy via the Java command line

```
c:\dev\projects>java -classpath .;c:/dev/tools/groovy/groovy-all-1.5.7.jar;
                  C:/dev/projects-2.0/jid3lib-0.5.jar
                  com.vanward.groovy.Songs c:\dev09\music\mp3s
Artist: U2 Album: Zooropa Song: Babyface
Artist: James Taylor Album: Greatest Hits Song: Carolina in My Mind
Artist: James Taylor Album: Greatest Hits Song: Fire and Rain
Artist: U2 Album: Zooropa Song: Lemon
Artist: James Taylor Album: Greatest Hits Song: Country Road
Artist: James Taylor Album: Greatest Hits Song: Don't Let Me
    Be Lonely Tonight
Artist: U2 Album: Zooropa Song: Some Days Are Better Than Others
Artist: Paul Simon Album: Graceland Song: Under African Skies
Artist: Paul Simon Album: Graceland Song: Homeless
Artist: U2 Album: Zooropa Song: Dirty Day
Artist: Paul Simon Album: Graceland Song: That Was Your Mother
```

## Embedding Groovy in Java code

While the command-line solution is easy and fun, it's not the be all, end all solution. If I were interested in a higher level of sophistication, I could import my MP3 song utility directly into my Java program. In this case, I'd import `Song.class` and use it as I would use any other class in the Java language. The classpath issues are the same as above: I need to make sure I include the *uber-Groovy* jar file, `Ant`, and the `jid3lib-0.5.jar` file. In Listing 5, you can see how I import my Groovy MP3 utility into a simple Java class:

### Listing 5. Embedded Groovy code

```
package com.vanward.gembed;

import com.vanward.groovy.Song;
import java.util.Collection;
import java.util.Iterator;

public class SongEmbedGroovy{

    public static void main(String args[]) {
        Collection coll = (Collection)Song.getSongsForDirectory
            ("C:\\music\\temp\\mp3s");
        for(Iterator it = coll.iterator(); it.hasNext();){
            System.out.println(it.next());
        }
    }
}
```

## Groovy classloaders

Just when you thought you'd learned it all, it turns out there are a few more ways to play with Groovy in the Java language. In addition to having options for how I will incorporate Groovy scripts into my Java programs through direct compilation, I also have some choices when it comes to how I will embed the scripts themselves.

For example, I could use Groovy's `GroovyClassLoader` to *dynamically* load a Groovy script and execute its behavior, as shown in Listing 6:

### Listing 6. `GroovyClassLoader` dynamically loads and executes a Groovy script

```
package com.vanward.gembed;

import groovy.lang.GroovyClassLoader;
import groovy.lang.GroovyObject;
import groovy.lang.MetaMethod;
import java.io.File;
```

```
public class CLEmbedGroovy{  
    public static void main(String args[]) throws Throwable{  
        ClassLoader parent = CLEmbedGroovy.class.getClassLoader();  
        GroovyClassLoader loader = new GroovyClassLoader(parent);  
  
        Class groovyClass = loader.parseClass(  
            new File("C:\\dev\\groovy-embed\\src\\groovy\\  
                com\\vanward\\groovy\\Songs.groovy"));  
  
        GroovyObject groovyObject = (GroovyObject)  
            groovyClass.newInstance();  
  
        Object[] path = {"C:\\music\\temp\\mp3s"};  
        groovyObject.setProperty("args", path);  
        Object[] argz = {};  
  
        groovyObject.invokeMethod("run", argz);  
    }  
}
```

### Meta me, baby

If you are one of those crazy cats who love reflection and the wonderful things you can do with it, then you'll go nuts over Groovy's *Meta*classes. Just like with reflection, using these classes, you can discover aspects about a *GroovyObject*, like its methods, and you can actually *create* new behavior and execute it. This, by the way, is the heart of Groovy -- and just imagine how it works when you're running scripts!

Note that by default, the class loader loads the class corresponding to the script name -- in this case *Songs.class*, *not* *Song.class*>. Because I (and you) know that *Songs.class* extends Groovy's *Script* class, it's a no-brainer that my next move is to execute the `run()` method.

You will recall that my Groovy script was also dependent on run-time arguments. So, I need to configure the `args` variable appropriately, as in this case I set the first element to a directory name.

## More dynamic options

An alternative to using compiled classes and dynamically loading *GroovyObjects* via classloaders is to use Groovy's nifty *GroovyScriptEngine* and the *GroovyShell* to dynamically execute Groovy scripts.

Embedding the *GroovyShell* object in your normal Java classes lets you dynamically execute Groovy scripts just as the classloader does. What's more, it gives you a number of options for how you will run your scripts. In Listing 7, you can see how *GroovyShell* is embedded in a normal Java class:

## Listing 7. Embedding the GroovyShell

```
package com.vanward.gembed;

import java.io.File;
import groovy.lang.GroovyShell;

public class ShellRunEmbedGroovy{

    public static void main(String args[]) throws Throwable{

        String[] path = {"C:\\music\\temp\\mp3s"};
        GroovyShell shell = new GroovyShell();
        shell.run(new File("C:\\dev\\groovy-embed\\src\\groovy\\
            com\\vanward\\groovy\\Songs.groovy"),
            path);
    }
}
```

As you can see, running a Groovy script is quite easy. I simply create an instance of `GroovyShell`, pass in the script name, and call the `run()` method.

But there's more. If you are so inclined, you can also ask a `GroovyShell` instance for the `Script` type of your script. With the `Script` type, you could then pass in a `Binding` object containing any run-time values and then proceed to call the `run()` method, as shown in Listing 8:

## Listing 8. Fun with the GroovyShell

```
package com.vanward.gembed;

import java.io.File;
import groovy.lang.Binding;
import groovy.lang.GroovyShell;
import groovy.lang.Script;

public class ShellParseEmbedGroovy{

    public static void main(String args[]) throws Throwable{
        GroovyShell shell = new GroovyShell();
        Script script = shell.parse(
            new File("C:\\dev\\groovy-embed\\src\\groovy\\
                com\\vanward\\groovy\\Songs.groovy"));

        Binding binding = new Binding();
        Object[] path = {"C:\\music\\temp\\mp3s"};
        binding.setVariable("args",path);
        script.setBinding(binding);

        script.run();
    }
}
```

## Groovy's script engine

The `GroovyScriptEngine` object works much like the `GroovyShell` for

dynamically running scripts. What's different about `GroovyScriptEngine` is that you can give it a series of directories upon instantiation and then have it evaluate multiple scripts on demand, as shown in Listing 9:

### Listing 9. The `GroovyScriptEngine` in action

```
package com.vanward.gembed;

import java.io.File;
import groovy.lang.Binding;
import groovy.util.GroovyScriptEngine;

public class ScriptEngineEmbedGroovy{

    public static void main(String args[]) throws Throwable{

        String[] paths = {"C:\\dev\\groovy-embed\\src\\groovy\\
            com\\vanward\\groovy"};
        GroovyScriptEngine gse = new GroovyScriptEngine(paths);
        Binding binding = new Binding();
        Object[] path = {"C:\\music\\temp\\mp3s"};
        binding.setVariable("args",path);

        gse.run("Songs.groovy", binding);
        gse.run("BusinessObjects.groovy", binding);
    }
}
```

In Listing 9, I pass an array containing my desired path into my instantiated `GroovyScriptEngine`, create the old familiar `Binding` object, and then execute the also familiar `Songs.groovy` script. Just for fun, I also execute the `BusinessObjects.groovy` script, which you may recall from the beginning of this discussion.

## The Bean Scripting Framework

Last, but certainly not least, is the venerable Bean Scripting Framework (BSF) from Jakarta. BSF attempts to offer a common API for embedding any scripting language in a normal Java application, including Groovy. This standard, but arguably least-common-denominator approach, allows you to embed a Groovy script effortlessly.

Remember the `BusinessObjects` script from earlier? In Listing 10, you can see how easily BSF lets me plug it into a normal Java program:

### Listing 10. BSF goes to work

```
package com.vanward.gembed;

import org.apache.bsf.BSFManager;
import org.codehaus.groovy.runtime.DefaultGroovyMethods;
import java.io.File;
```

```
import groovy.lang.Binding;

public class BSFEmbedGroovy{

    public static void main(String args[]) throws Exception {
        String fileName = "C:\\dev\\project\\src\\groovy\\
            com\\vanward\\groovy\\BusinessObjects.groovy";
        //this is required for bsf-2.3.0
        //the "groovy" and "gy" are extensions
        BSFManager.registerScriptingEngine("groovy",
            "org.codehaus.groovy.bsf.GroovyEngine", new
            String[] { "groovy" });
        BSFManager manager = new BSFManager();
        //DefaultGroovyMethods.getText just returns a
        //string representation of the contents of the file
        manager.exec("groovy", fileName, 0, 0,
            DefaultGroovyMethods.getText(new File(fileName)));
    }
}
```

## Conclusion

If one thing is clear from this article, it's that Groovy presents a myriad of options for reuse inside your Java code. From compiling Groovy scripts into plain old Java .class files to dynamically loading and running scripts, the key aspects to consider are flexibility and coupling. Compiling Groovy scripts into normal .class files is the simplest choice for *using* the functionality you are trying to embed, but dynamically loading the scripts makes it easier to add or modify behavior without sacrificing time on compilation. (Of course, this option only works if the interface doesn't change.)

Embedding a scripting language in normal Java isn't an everyday occurrence, but opportunities do present themselves from time to time. In the examples I presented here, I embedded my simple directory-search utility in a Java-based application that could easily become an MP3 player application or some other MP3 utility. While I *could* have rewritten my MP3 file finder utility in Java code, I had no need to: Groovy is perfectly compatible with the Java language, and besides, I had lots of fun tinkering with all the options!

## Downloads

Description	Name	Size	Download method
Sample code	j-pg12144.zip	7KB	<a href="#">HTTP</a>

[Information about download methods](#)

## About the authors

### Andrew Glover

Andrew Glover is a developer, author, speaker, and entrepreneur. He is the founder of the [easyb](#) Behavior-Driven Development (BDD) framework and is the co-author of three books: [Continuous Integration](#), [Groovy in Action](#), and [Java Testing Patterns](#). He teaches a wide variety of Groovy-, Grails-, and testing-related classes at [ThirstyHead.com](#). You can keep up with Andy at [thediscoblog.com](#) where he routinely blogs about software development.

---

### Scott Davis

Scott Davis is an internationally recognized author, speaker, and software developer. He is the founder of [ThirstyHead.com](#), a Groovy and Grails training company. His books include [Groovy Recipes: Greasing the Wheels of Java](#), [GIS for Web Developers: Adding Where to Your Application](#), [The Google Maps API](#), and [JBoss At Work](#). He writes two ongoing article series for IBM developerWorks: [Mastering Grails](#) and [Practically Groovy](#).